Repetition Via Loops

In which our hero learns how to create a loop, but is disappointed: it’s not music, but rather code, that is doing the looping.

12.1 To Repeat Oneself

One of the elements of successful teaching is to repeat the important things you are saying. This is the key: if you have something essential to get across, you must, as an educator, say it over and over again. So yes, throughout this book, we will describe what you should know and emphasize it repeatedly. And again. And then, as they say in famous Broadway musicals, “one last time” [H15].

Computer programs often have to repeat themselves too. However, it’s not for the sake of emphasis or because the computer program is teaching anyone anything. It’s that sometimes it is useful to apply the same bit of code to a different piece of data, over and over again. We call such repetition looping; creating such a loop is thus commonplace in many programming languages, as it is in C.

As was the case with the if statement, loops alter the flow of control of your program, and thus are another important way to execute code in non-straightline order. Thus, with each new construct, make sure you understand how the C instruction pointer advances.

C offers three looping constructs: while, for, and do-while. All are essentially equivalent – you can certainly do with one what you can do with the others – but sometimes it is slightly more concise to use one specific variant. Thus, our focus: how can we create loops in C?

12.2 Erst, while

The most basic construct for repetition is the while loop. Here is a simple loop that multiplies a number (sum) by two until the value of sum is no longer less than 100:
int sum = 1;
while (sum < 100) {
    sum = sum * 2;
}

The code runs as follows. First, \texttt{sum} is assigned the value 1. Then, the execution of the program encounters the \texttt{while}, meaning that an \textbf{expression} (between the mandatory parentheses) is next to be evaluated. In this case, \texttt{sum<100} evaluates to 1 (true), and thus, the control flow will enter the \textbf{loop body}; therein, the code will multiply \texttt{sum} by 2 and store the result back into \texttt{sum}.

This process continues, setting \texttt{sum} equal to 4, 8, 16, 32, 64, and finally 128. At that point, the expression is evaluated one last time [H15] and finally evaluates to 0 (false). At this point, the execution skips over the loop body, and would execute whatever line of code immediately follows the loop body.

The generic form of \texttt{while} is thus as follows:

\begin{verbatim}
while (expression) {
    statement_1;
    statement_2;
    ...
    statement_n;
}
\end{verbatim}

There is also a short form, again without curly braces:

\begin{verbatim}
while (expression)
    statement_1;
\end{verbatim}

Here is another simple example. In this case, the code just sums up the numbers from 1 to 10$^1$.

\begin{verbatim}
int sum = 0;
int i = 1;
while (i <= 10) {
    sum = sum + i;
    i = i + 1;
}
\end{verbatim}

\footnote{Yes, there is an easier way to do this; the sum of numbers from 1 to $n$ is equal to \((n + 1) \cdot \frac{n}{2}\) (see if you can reason out why, if you don't know already). As the legend has it, young Carl Gauss figured this shortcut out as a way to avoid the busy work his teachers had given him [NPP06].}
12.3 Infinite Loops and \texttt{break}

Some loops keep going and going and going. We call them infinite loops. Can we really be sure they are infinite? Well, you can, by just running them and seeing if they stop. If they stop, they are definitely not infinite. Otherwise, well, they just keep running. Here is how you write an infinite loop with \texttt{while}:

```
while (1)
{

}
```

As you can see, the expression in this case is just the value 1 (true), which means the code will keep executing the loop body. You don’t often see code like the above, but it does exist in some places\(^2\).

There are also times when you might wish to break out of a while loop, instead of waiting for the expression to evaluate. To do so, just use the \texttt{break} keyword. Here is a contrived example of a loop which increments a counter until it is greater than 10, and then breaks out of the loop; the \texttt{break} jumps the code to the first statement past the end of the loop body.

```
int i = 0;
while (1) {
    if (i > 10)
        break;
    i = i + 1;
}
```

12.4 Sometimes You Should Just \texttt{continue}

When looping, sometimes you’ll want to stop running in the middle of the body of the loop and go back to the top, evaluate the expression, and then continue looping (or not, if the expression is false). The \texttt{continue} keyword lets you do this. Let’s say you wanted to add up the numbers from 1 to 10 again, but (for some reason) wanted to skip over 3. This code would do exactly that.

```
int i = 1;
while (i <= 10) {
    i = i + 1;
    if (i == 3)
        continue;
    sum = sum + i;
}
```

\(^2\)For example, in the dark corners of old operating systems, when the machine had nothing to do, the system scheduler would drop into a loop like this (called the \texttt{idle loop}); only via an \texttt{interrupt} (which forces the CPU to jump to some other code) would the spell of infinity be broken [AD17].
When the CAM encounters the `continue` statement, it jumps back up to the expression evaluation, avoiding (in this case) the line where `sum` is incremented. In many cases, you could construct this same behavior with `if` statements in the loop body, but `continue` can make for cleaner and more concise code.

12.5 Another Looping Construct: `for`

A second looping construct is available in C, the `for` loop. Here is a simple loop that uses `for`:

```c
int sum = 0;
int i;
for (i = 0; i < 10; i = i + 1) {
    sum = sum + i;
}
```

Because you already understand `while` loops, we can recast this code to an exact equivalent using `while`:

```c
int sum = 0;
int i;
i = 0; // initial statement
while (i < 10) {
    sum = sum + i;
    i = i + 1; // post-body statement
}
```

The more generic form for `for` is thus:

```c
for (initial statement; expression; post-body statement) {
    statement_1;
    statement_2;
    ...;
    statement_n;
}
```

As you may have concluded, the `initial statement` is executed exactly once, before anything else with the looping takes place. Then, the expression is evaluated, to determine whether to enter the loop body. If the loop body is entered, each statement within is executed; when the end of the body is reached, the `post-body statement` is executed. At this point, the code loops back to the expression evaluation portion again, and the cycle repeats.

These `for` loops can be quite convenient and thus often are used instead of `while`. One specific place is when a loop variable of some kind is being initialized once and incremented every iteration; thus, it is common to see some code like this:
for (i = 0; i < 10; i++) {
    ...
}

We should also note that each part of the for parenthetical clause is optional (where the semi-colons are not). Thus, the following is perfectly legal (with no initial statement or post-body statement):

    i = 0;
    for ( ; i < 10; ) {
        sum += i;
        i++;
    }

You can even write an infinite loop by leaving out the expression component (while, in contrast, requires something):

    for (;;) {
        ;
    }

12.6 A New Operator: The Comma

Sometimes, you'd like to initialize more than one variable in a for loop initial statement; similarly, sometimes you'd like to increment two different variables in the post-body statement. C allows you to do this, but to do so, you need to use a new operator, the comma operator (,). Here is an example:

    for (i = 0, j = 1; i < 10; i++, j++) {
        ...
    }

The comma operator is a binary operator. It first evaluates the expression on the left of the comma, and then discards the results; after, it evaluates the expression on the right, and that value is the value of the entire expression. You can also use it elsewhere in your code, like this:

    x = 3, y = 4;

This expression assigns 3 to x, and 4 to y. If you are paying close attention, you will have concluded that the precedence of the comma operator is low, even lower than =, and you would be right (try thinking about evaluating the above expression if the opposite were true). See Figure 12.1 for details.
You might also be thinking, why not just use semi-colons instead, like this?

```c
x = 3; y = 4;
```

And you would also be right, at least in this case. The reason that the more standard usage of semi-colon to separate separate assignments doesn’t work in the `for` loop above, though, is that `for` expects exactly two semi-colons within the parentheses; hence, you need to do something else to include multiple expressions in the initial statement.

Quick quiz: given the operation of the comma, what would `x` be after each of the following two statements?

```c
x = 3, 4;
```

```c
x = (3, 4);
```

The answer in the first case is 3, because comma has lower precedence than assignment; in the second case, it’s 4, because the parentheses group 3, 4, which evaluates to the expression on the right of the comma (4). In both cases, when you try to write code like this, C should warn you that you’re doing something a little odd.

### 12.7 One Last Construct: do ... while

One last looping construct is the dreaded `do ... while`. It looks like this, and actually, is not really dreaded by anybody:

```c
do {
    statement_1;
    statement_2;
    ...
    statement_n;
} while (expression);
```

We could write an equivalent piece of code with `while`, and it would look like this:
statement_1;
statement_2;
...
statement_n;
while (expression) {
  statement_1;
  statement_2;
  ...
  statement_n;
}

As you can see, the **do-while loop** always executes one time, and only after evaluates the expression to see whether it should continue. Having this construct in our bag of tricks avoids repeating all of that code in the cases where you need to ensure at least one pass through the body of the loop.

### 12.8 Summary

We have seen a huge number of ways to repeat code in a loop. Repetition is the key to many aspects of computing, and thus looping, in imperative languages like C, is a workhorse. Learn to use loops and you can conquer the world, or, minimally, write a few short lines of code that do something pretty useful.
References

Lin-Manuel Miranda  
Premiered in 2015 at the Public Theater  
Yes, we have been itching to get in a Hamilton reference. Really, the most amazing piece of art we’ve seen in many years. Most people can’t afford tickets, but definitely can afford the cast album, which is like $20 and probably the best $20 you’ll spend this year. This particular reference, by the way, is to “One Last Time”, or George Washington’s farewell song. Originally, it was called “One Last Ride” which you can still find a copy of on the Internet if you look for it.

[NPP06] “What is the sum of the first 100 whole numbers?”  
Natasha, Paul, and Penny  
mathcentral.uregina.ca/qq/database/qq.02.06/jo1.html  
Amazingly this is the first website that turned up in a Google search for the details of Gauss and his precocious calculations. Who Natasha, Paul, and Penny are remains a mystery.